18 Improving your application's security

This chapter covers

- Encrypting traffic using HTTPS and configuring local SSL certificates
- Defending against cross-site scripting attacks
- Protecting from cross-site request forgery attacks
- Allowing calls to your API from other apps using CORS

Web application security is a hot topic at the moment. Practically every week another breach is reported, or confidential details are leaked. It may seem like the situation is hopeless, but the reality is that the vast majority of breaches could have been avoided with the smallest amount of effort.

In this chapter we'll look at a few different ways to protect your application and your application's users from attackers. Because security is an extremely broad topic that covers lots of different avenues, this chapter is by no means an exhaustive guide. It's intended to make you aware of some of the most common threats to your app and how to counteract them, and also to highlight areas where you can inadvertently introduce vulnerabilities if you're not careful.

TIP I strongly advise exploring additional resources around security after you've read this chapter. The Open Web Application Security Project (OWASP) (www.owasp.org) is an excellent resource, though it can be a little dry. Alternatively, Troy Hunt has some excellent courses and workshops on security, geared toward .NET developers (www.troyhunt.com/).

We'll start by looking at how to add HTTPS encryption to your website so that users can access your app without the risk of third parties spying on or modifying the content as it travels over the internet. This is effectively mandatory for production apps these days, and it is heavily encouraged by the makers of modern browsers such as Chrome and Firefox. You'll see how to use the ASP.NET Core development certificate to use HTTPS locally, how to configure an app for HTTPS in production, and how to enforce HTTPS across your whole app.

In sections 18.2 and 18.3 you'll learn about two potential attacks that should be on your radar: cross-site scripting (XSS) and cross-site request forgery (CSRF). We'll explore how the attacks work and how you can prevent them in your apps. ASP.NET Core has built-in protection against both types of attack, but you have to remember to use the protection correctly and resist the temptation to circumvent it, unless you're certain it's safe to do so.

Section 18.4 deals with a common scenario—you have an application that wants to use JavaScript AJAX (Asynchronous JavaScript and XML) requests to retrieve data from a second app. By default, web browsers block requests to other apps, so you need to enable cross-origin resource sharing (CORS) in your API to achieve this. We'll look at how CORS works, how to create a CORS policy for your app, and how to apply it to specific action methods.

The final section of this chapter, section 18.5, covers a collection of common threats to your application. Each one represents a potentially critical flaw that an attacker could use to compromise your application. The solutions to each threat are generally relatively simple; the important thing is to recognize where the flaws could exist in your own apps so that you can ensure that you don't leave yourself vulnerable.

We'll start by looking at HTTPS and why you should use it to encrypt the traffic between your users' browsers and your app. Without HTTPS, attackers could subvert many of the safeguards you add to your app, so it's an important first step to take.

18.1 Adding HTTPS to an application

In this section you'll learn about HTTPS: what it is, and why you need to be aware of it for all your production applications. You'll see two approaches to adding HTTPS to your application: supporting HTTPS directly in your application and using SSL/TLS-offloading with a reverse proxy. You'll then learn how to use the development certificate to work with HTTPS on your local machine, and how to add an HTTPS certificate to your app in production. Finally, you'll learn how to enforce HTTPS in your app using best practices such as security headers and HTTP redirection.

So far in this book, I've shown how the user's browser sends a request across the internet to your app using the HTTP protocol. We haven't looked too much into the details of that protocol, other than to establish that it uses *verbs* to describe the type of request (such as `GET` and `POST`), that it contains *headers* with metadata about the request, and optionally includes a *body* payload of data.

By default, HTTP requests are unencrypted; they're plain text files being sent over the internet. Anyone on the same network as a user (such as someone using the same public Wi-Fi in a coffee shop) can read the requests and responses sent back and forth. Attackers can even *modify* the requests or responses as they're in transit.

Using unencrypted web apps in this way presents both a privacy and a security risk to your users. Attackers could read the data sent in forms and returned by your app, inject malicious code into your responses to attack users, or steal authentication cookies and impersonate the user on your app.

To protect your users, your app should encrypt the traffic between the user's browser and your app as it travels over the network by using the HTTPS protocol. This is similar to HTTP traffic, but it uses an SSL/TLS[1] certificate to encrypt requests and responses, so attackers cannot read or modify the contents. In browsers, you can tell that a site is using HTTPS by the https:// prefix to URLs (notice the "s"), or sometimes, by a padlock, as shown in figure 18.1.

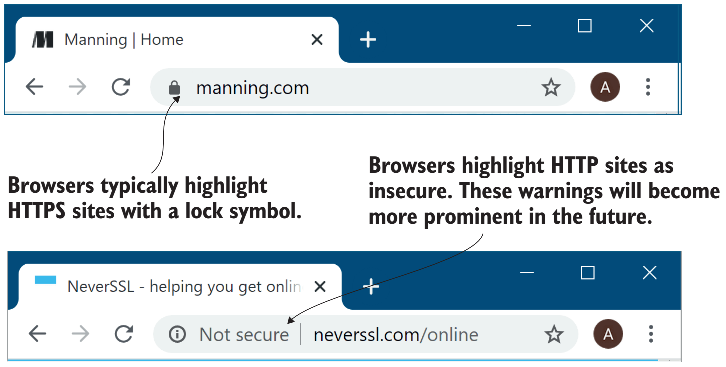

Manning | Home
manning.com
Browsers typically highlight HTTPS sites with a lock symbol.
Browsers highlight HTTP sites as insecure. These warnings will become more prominent in the future.
NeverSSL - helping you get onlin
Not secure
neverssl.com/online

Figure 18.1 Encrypted apps using HTTPS and unencrypted apps using HTTP in Edge. Using HTTPS protects your application from being viewed or tampered with by attackers.

TIP For details on how the SSL/TLS protocols work, see chapter 9 of *Real-World Cryptography* by David Wong (Manning, 2021), http://mng.bz/zxz1.

[1] SSL is an older standard that facilitates HTTPS, but the SSL protocol has been superseded by Transport Layer Security (TLS), so I'll be using TLS preferentially throughout this chapter.

The reality is that, these days, you should always serve your production websites over HTTPS. The industry is pushing toward HTTPS by default, with most browsers moving to mark HTTP sites as explicitly "not secure." Skipping HTTPS will hurt the perception of your app in the long run, so even if you're not interested in the security benefits, it's in your best interest to set up HTTPS.

To enable HTTPS you need to obtain and configure a TLS certificate for your server. Unfortunately, although that process is a lot easier than it used to be, and it's now essentially free thanks to Let's Encrypt (https://letsencrypt.org/), it's still far from simple in many cases. If you're setting up a production server, I recommend carefully following the tutorials on the Let's Encrypt site. It's easy to get it wrong, so take your time.

> **TIP** If you're hosting your app in the cloud, most providers will provide one-click TLS certificates so that you don't have to manage certificates yourself. This is *extremely* useful, and I highly recommend it for everyone.[2]

As an ASP.NET Core application developer, you can often get away without *directly* supporting HTTPS in your app by taking advantage of the reverse-proxy architecture, as shown in figure 18.2, in a process called SSL/TLS offloading/termination. Instead of your application handling requests using HTTPS directly, it continues to use HTTP. The reverse proxy is responsible for encrypting and decrypting HTTPS traffic to the browser. This often gives you the best of both worlds—data is encrypted between the user's browser and the server, but you don't have to worry about configuring certificates in your application.[3]

Depending on the specific infrastructure where you're hosting your app, SSL/TLS could be offloaded to a dedicated device on your network, a third-party service like Cloudflare, or a reverse proxy (such as IIS, NGINX, or HAProxy) running on the same or a different server.

Nevertheless, in some situations, you may need to handle SSL/TLS directly in your app:

- *If you're exposing Kestrel to the internet directly, without a reverse proxy.* This became more common with ASP.NET Core 3.0 due to hardening of the Kestrel server. It is also often the case when you're developing your app locally.
- *If having HTTP between the reverse proxy and your app is not acceptable.* While securing traffic *inside* your network is less critical compared to external traffic, it is undoubtedly more secure to use HTTPS for internal traffic too.
- *If you're using technology that requires HTTPS.* Some newer network protocols, such as gRPC and HTTP/2 require an HTTPS connection.

---

[2] You don't even have to be hosting your application in the cloud to take advantage of this. Cloudflare (www.cloudflare.com) provides a CDN service that you can add TLS to. You can even use it for free.

[3] If you're concerned that the traffic is unencrypted between the reverse proxy and your app, then I recommend reading Troy Hunt's "CloudFlare, SSL and unhealthy security absolutism" post: http://mng.bz/eHCi. It discusses the pros and cons of the issue as it relates to using Cloudflare to provide HTTPS encryption.

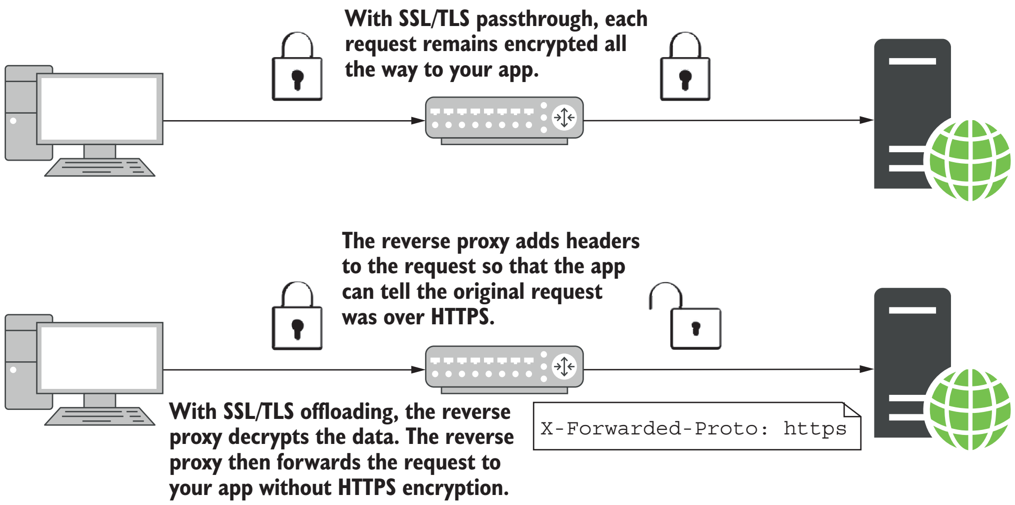

With SSL/TLS passthrough, each request remains encrypted all the way to your app.
The reverse proxy adds headers to the request so that the app can tell the original request was over HTTPS.
With SSL/TLS offloading, the reverse proxy decrypts the data. The reverse proxy then forwards the request to your app without HTTPS encryption.
X-Forwarded-Proto: https

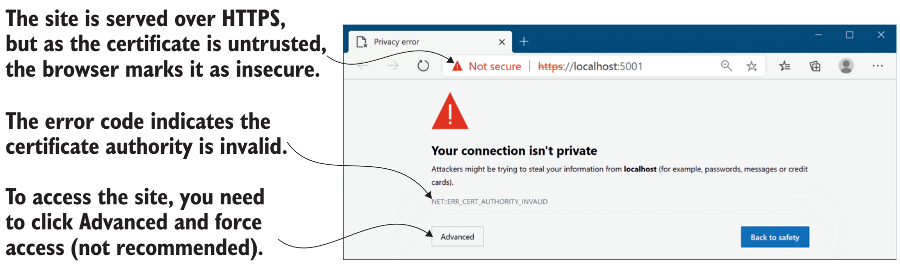
Figure 18.2 You have two options when using HTTPS with a reverse proxy: SSL/TLS passthrough and SSL/TLS offloading. In SSL/TLS passthrough, the data is encrypted all the way to your ASP.NET Core app. For SSL/TLS offloading, the reverse proxy handles decrypting the data, so your app doesn't have to.

In each of these scenarios, you'll need to configure a TLS certificate for your application so Kestrel can receive HTTPS traffic. In section 18.1.1 you'll see the easiest way to get started with HTTPS when developing locally, and in section 18.1.2 you'll see how to configure your application for production.

18.1.1 Using the ASP.NET Core HTTPS development certificates

Working with HTTPS certificates is easier than it used to be, but unfortunately it can still be a confusing topic, especially if you're a newcomer to the web. The .NET SDK, Visual Studio, and IIS Express try to improve this experience by handling a lot of the grunt-work for you.

The first time you run a `dotnet` command using the .NET SDK, the SDK installs an HTTPS development certificate onto your machine. Any ASP.NET Core application you create using the default templates (or for which you don't explicitly configure certificates) will use this development certificate to handle HTTPS traffic. However, the development certificate is not *trusted* by default. That means you'll get a browser warning, as shown in figure 18.3 when accessing a site after first installing the .NET SDK.


The site is served over HTTPS, but as the certificate is untrusted, the browser marks it as insecure.
The error code indicates the certificate authority is invalid.
To access the site, you need to click Advanced and force access (not recommended).
Privacy error
Not secure
~~https~~://localhost:5001
Advanced
Back to safety

Your connection isn't private

Attackers might be trying to steal your information from **localhost** (for example, passwords, messages or credit cards).

NET::ERR_CERT_AUTHORITY_INVALID

Figure 18.3 The developer certificate is not trusted by default, so apps serving HTTPS traffic using it will be marked as insecure by browsers. Although you can bypass the warnings if necessary, you should instead update the certificate to be trusted.

A brief primer on certificates and signing

HTTPS uses *public key cryptography* as part of the data-encryption process. This uses two keys: a *public* key that *anyone* can see, and a *private* key that only *your* server can see. Anything encrypted with the public key can only be decrypted with the private key. That way, a browser can encrypt something with your server's public key, and only your server can decrypt it. A complete TLS certificate consists of both the public and private parts.

When a browser connects to your app, the server sends the public key part of the TLS certificate. But how does the browser know that it was definitely *your* server that sent the certificate? To achieve this, your TLS certificate contains additional certificates, including a certificate from a third party, a certificate authority (CA). This trusted certificate is called a *root certificate*.

CAs are special trusted entities, and browsers are hardcoded to trust certain root certificates. In order for the TLS certificate for your app to be trusted, it must contain (or be signed by) a trusted root certificate.

When you use the ASP.NET Core development certificate, or if you create your own self-signed certificate, your site's HTTPS is missing that trusted root certificate. That means browsers won't trust your certificate and won't connect to your server by default. To get around this, you need to tell your development machine to explicitly trust the certificate.

In production, you can't use a development or self-signed certificate, as a user's browser won't trust it. Instead, you need to obtain a signed HTTPS certificate from a service like Let's Encrypt, or from a cloud provider like AWS, Azure, or Cloudflare. These certificates will already be signed by a trusted CA, so they will be automatically trusted by browsers.

To solve these browser warnings, you need to *trust* the certificate. Trusting a certificate is a sensitive operation; it's saying, "I know this certificate doesn't look quite right, but just ignore that," so it's hard to do automatically. If you're running on Windows or macOS, you can trust the development certificate by running

```
dotnet dev-certs https --trust
```

This command trusts the certificate by registering it in the operating system's "certificate store." After you run this command, you should be able to access your websites without seeing any warnings or "not secure" labels, as shown in figure 18.4.

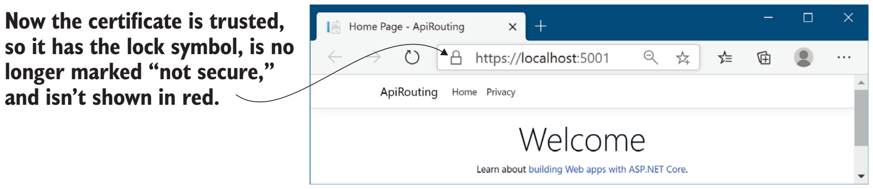

Now the certificate is trusted, so it has the lock symbol, is no longer marked "not secure," and isn't shown in red.
Home Page - ApiRouting
https://localhost:5001
ApiRouting
Home
Privacy
Welcome
Learn about building Web apps with ASP.NET Core.

Figure 18.4 Once the development certificate is trusted, you will no longer see browser warnings about the connection.

TIP You may need to close your browser after trusting the certificate to clear the browser's cache.

The developer certificate works smoothly on Windows and macOS. Unfortunately, trusting the certificate in Linux is a little trickier and depends on the particular flavor you're using. On top of that, software on Linux often uses its own certificate store, so you'll probably need to add the certificate directly to your favorite browser. I suggest looking at the documentation for your favorite browser to figure out the best approach. For advice on other platforms, such as Docker, see Microsoft's "How to set up a developer certificate for Docker" section in the "Enforce HTTPS in ASP.NET Core" documentation: http://mng.bz/0mBJ.

If you're using Windows, Visual Studio, and IIS Express for development, then you may not find the need to trust the development certificate. IIS Express acts as a reverse proxy when you're developing locally, so it handles the SSL/TLS setup itself. On top of that, Visual Studio should trust the IIS development certificate as part of installation, so you may never see the browser warnings at all.

The ASP.NET Core and IIS development certificates make it easy to use Kestrel with HTTPS locally, but those certificates won't help once you move to production. In the next section I'll show you how to configure Kestrel to use a production TLS certificate.

18.1.2 Configuring Kestrel with a production HTTPS certificate

Creating a TLS certificate for production is often a laborious process, as it requires proving to a third-party certificate authority (CA) that you own the domain you're creating the certificate for. This an important step in the "trust" process and ensures that attackers can't impersonate your servers. The result of the process is one or more files, which is the HTTPS certificate you need to configure for your app.

> **TIP** The specifics of how to obtain a certificate vary by provider and by your OS platform, so follow your provider's documentation carefully. The vagaries and complexities of this process are one of the reasons I strongly favor the SSL/TLS-offloading or "one-click" approaches described previously. Those approaches mean my apps don't need to deal with certificates, and I don't need to use the approaches described in this section; I delegate that responsibility to another piece of the network, or to the underlying platform.

Once you have a certificate, you need to configure Kestrel to use it to serve HTTPS traffic. In chapter 16 you saw how to set the port your application listens on with the `ASPNETCORE_URLS` environment variable or via the command line, and you saw that you could provide an HTTPS URL. As you didn't provide any certificate configuration, Kestrel used the development certificate by default. In production you need to tell Kestrel which certificate to use.

Kestrel is very configurable, allowing you to configure your certificates in multiple ways. You can use different certificates for different ports, you can load from a .pfx file or from the OS certificate store, or you can have a different configuration for each URL endpoint you expose. For full details, see the "Endpoint configuration" section in Microsoft's "Kestrel web server implementation in ASP.NET Core" documentation: http://mng.bz/KMdX.

The following listing shows one possible way to set a custom HTTPS certificate for your production app, by configuring the default certificate Kestrel uses for HTTPS connections. You can add the `"Kestrel:Certificates:Default"` section to your appsettings.json file (or using any other configuration source, as described in chapter 11) to define the .pfx file of the certificate to use. You must also provide the password for accessing the certificate.

Listing 18.1 Configuring the default HTTPS certificate for Kestrel using a .pfx file

```
{
  "Kestrel": {                              Create a configuration section
    "Certificates": {                       at Kestrel:Certificates:Default.
      "Default": {
        "Path": "localhost.pfx",        <-- The relative or absolute
        "Password": "testpassword"      <-- path to the certificate
      }
    }                                       The password for
  }                                         opening the certificate
}
```

The preceding example is the simplest way to replace the HTTPS certificate, as it doesn't require changing any of Kestrel's defaults. You can use a similar approach to load the HTTPS certificate from the OS certificate store (on Windows or macOS), as shown in the "Endpoint configuration" documentation mentioned previously (http://mng.bz/KMdX).

WARNING Listing 18.1 has hardcoded the certificate filename and password for simplicity, but you should either load these from a configuration store like user-secrets, as you saw in chapter 11, or load the certificate from the local store. *Never* put production passwords in your appsettings.json files.

All the default ASP.NET Core templates configure your application to serve both HTTP and HTTPS traffic, and with the configuration you've seen so far, you can ensure your application can handle both HTTP and HTTPS in development and in production.

However, whether you use HTTP or HTTPS may depend on the URL users click when they first browse to your app. For example, if your app listens using the default URLs, http://localhost:5000 for HTTP traffic and https://localhost:5001 for HTTPS traffic, if a user navigates to the HTTP URL, their traffic will be unencrypted. Seeing as you've gone to all the trouble to set up HTTPS, it's probably best that you force users to use it.

18.1.3 Enforcing HTTPS for your whole app

Enforcing HTTPS across your whole website is practically required these days. Browsers are beginning to explicitly label HTTP pages as insecure; for security reasons you *must* use TLS any time you're transmitting sensitive data across the internet, and, thanks to HTTP/2, adding TLS can *improve* your app's performance.[4]

There are multiple approaches to enforcing HTTPS for your application. If you're using a reverse proxy with SSL/TLS-offloading, it might be handled for you anyway, without having to worry about it within your apps. Nevertheless, it doesn't hurt to enforce SSL/TLS in your applications too, regardless of what the reverse proxy may be doing.

NOTE If you're building a Web API, rather than a Razor Pages app, it's common to just reject HTTP requests, without using the approaches described in this section. These protections apply primarily when building apps to be consumed in a browser. For more details, see Microsoft's "Enforce HTTPS in ASP.NET Core" documentation: http://mng.bz/j46a.

One approach to improving the security of your app is to use HTTP *security headers*. These are HTTP headers sent as part of your HTTP response that tell the browser how

[4] HTTP/2 offers many performance improvements over HTTP/1.x, and all modern browsers require HTTPS to enable it. For a great introduction to HTTP/2, see Google's "Introduction to HTTP/2": http://mng.bz/9M8j.

it should behave. There are many different headers available, most of which restrict the features your app can use in exchange for increased security.[5] In the next chapter you'll see how to add your own custom headers to your HTTP responses by creating custom middleware.

One of these security headers, the HTTP Strict Transport Security (HSTS) header, can help ensure browsers use HTTPS where it's available, instead of defaulting to HTTP.

Enforcing HTTPS with HTTP Strict Transport Security headers

It's unfortunate, but by default, browsers always load apps over HTTP, unless otherwise specified. That means your apps typically must support both HTTP and HTTPS, even if you don't want to serve any traffic over HTTP. One mitigation for this (and a security best practice), is to add HTTP Strict Transport Security headers to your responses.

> **DEFINITION** HTTP Strict Transport Security (HSTS) is a header that instructs the browser to use HTTPS for all *subsequent* requests to your application. The browser will no longer send HTTP requests to your app and will only use HTTPS instead. It can only be sent with responses to HTTPS requests. It is only relevant for requests originating from a browser—it has no effect on server-to-server communication.

HSTS headers are strongly recommended for *production* apps. You generally don't want to enable them for local development, as that would mean you could never run a non-HTTPS app locally. In a similar fashion, you should only use HSTS on sites for which you *always* intend to use HTTPS, as it's hard (sometimes impossible) to turn-off HTTPS once it's enforced with HSTS.

ASP.NET Core comes with built-in middleware for setting HSTS headers, which is included in some of the default templates automatically. The following listing shows how you can configure the HSTS headers for your application using the `HstsMiddleware` in Startup.cs.

Listing 18.2 Using `HstsMiddleware` to add HSTS headers to an application

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
        services.AddHsts(options =>
        {
            options.MaxAge = TimeSpan.FromHours(1);
        });
    }
```

Configure your HSTS header settings. This changes the MaxAge from the default of 30 days.

[5] Scott Helme has some great guidance on this and other security headers you can add to your site, such as the Content Security Policy (CSP) header. See "Hardening your HTTP response headers" on his website: https://scotthelme.co.uk/hardening-your-http-response-headers/.

```
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsProduction())
        {
            app.UseHsts();
        }

        app.UseStaticFiles();
        app.UseRouting();
        app.UseAuthorization();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

You shouldn't use HSTS in local environments.

Adds the HstsMiddleware

The HstsMiddleware should be very early in the middleware pipeline.

> **TIP** The preceding example shows how to change the `MaxAge` sent in the HSTS header. It's a good idea to start with a small value initially. Once you're sure your app's HTTPS is functioning correctly, increase the age for greater security. For more details on HSTS see Scott Helme's article, "HSTS—The missing link in Transport Layer Security": https://scotthelme.co.uk/hsts-the-missing-link-in-tls/.

HSTS is a great option for forcing users to use HTTPS on your website. But one problem with the header is that it can only be added to HTTPS requests. That means you must have *already* made an HTTPS request before HSTS kicks in: if the *initial* request is HTTP, no HSTS header is sent, and you *stay* on HTTP! That's unfortunate, but you can mitigate it by redirecting insecure requests to HTTPS immediately.

REDIRECTING FROM HTTP TO HTTPS WITH THE HTTPS REDIRECTION MIDDLEWARE

The `HstsMiddleware` should generally be used in conjunction with middleware that redirects all HTTP requests to HTTPS.

> **TIP** It's possible to apply HTTPS redirection to only parts of your application, such as to specific Razor Pages, but I don't recommend that, as it's too easy to open up a security hole in your application.

ASP.NET Core comes with `HttpsRedirectionMiddleware`, which you can use to enforce HTTPS across your whole app. You add it to the middleware pipeline in the `Configure` section of `Startup`, and it ensures that any requests that pass through it are secure. If an HTTP request reaches the `HttpsRedirectionMiddleware`, the middleware immediately short-circuits the pipeline with a redirect to the HTTPS version of the request. The browser will then repeat the request using HTTPS instead of HTTP.

> **NOTE** The eagle-eyed among you will notice that even with the HSTS and redirection middleware, there is still an inherent weakness. By default, browsers will always make an initial, *insecure*, request over HTTP to your app. The only way to avoid this is by HSTS-preloading, which tells browsers to *always* use HTTPS. You can find a great guide to HSTS, including preloading, on the

ForwardPMX site: “The Ultimate Guide to HSTS Protocol” by Chris Herbrand, http://mng.bz/Wdmg.

The `HttpsRedirectionMiddleware` is added in the default ASP.NET Core templates. It is typically placed after the error handling and `HstsMiddleware`, as shown in the following listing. By default, the middleware redirects all HTTP requests to the secure endpoint, using an HTTP `307 Temporary Redirect` status code.

Listing 18.3 Using `HttpsRedirectionMiddleware` to enforce HTTPS for an application

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
        app.UseExceptionHandler("/Error");
        if (env.IsProduction())
        {
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();
        app.UseAuthorization();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

Adds the HttpsRedirectionMiddleware to the pipeline. Redirects all HTTP requests to HTTPS.

The `HttpsRedirectionMiddleware` will automatically redirect HTTP requests to the first configured HTTPS endpoint for your application. If your application isn’t configured for HTTPS, the middleware *won’t* redirect and instead will log a warning:

```
warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
      Failed to determine the https port for redirect.
```

If you want the middleware to redirect to a different port than Kestrel knows about, you can configure that by setting the `ASPNETCORE_HTTPS_PORT` environment variable. This is sometimes necessary if you’re using a reverse proxy, and it can be set in alternative ways, as described in Microsoft’s “Enforce HTTPS in ASP.NET Core” documentation: http://mng.bz/QmN4.

SSL/TLS offloading, header forwarding, and detecting secure requests

At the start of section 18.1, I encouraged you to consider terminating HTTPS requests at a reverse proxy. That way, the user uses HTTPS to talk to the reverse proxy, and the reverse proxy talks to your app using HTTP. With this setup, your users are protected but your app doesn’t have to deal with TLS certificates itself.

(continued)

In order for the `HttpsRedirectionMiddleware` to work correctly, Kestrel needs some way of knowing whether the *original* request that the reverse proxy received was over HTTP or HTTPS. The reverse proxy communicates to your app over HTTP, so Kestrel can't figure that out without extra help.

The standard approach used by most reverse proxies (such as IIS, NGINX, and HAProxy) is to add headers to the request before forwarding it to your app. Specifically, a header called `X-Forwarded-Proto` is added, indicating whether the original request protocol was HTTP or HTTPS.

ASP.NET Core includes `ForwardedHeadersMiddleware` to look for this header (and others) and update the request accordingly, so your app treats a request that was *originally* secured by HTTPS as secure for all intents and purposes.

If you're using IIS with the `UseIisIntegration()` extension, the header forwarding is handled for you automatically. If you're using a different reverse proxy, such as NGINX or HAProxy, you can enable the middleware by setting the environment variable `ASPNETCORE_FORWARDEDHEADERS_ENABLED=true`, as you saw in chapter 16. Alternatively, you can manually add the middleware to your application, as shown in section 16.3.2.

When the reverse proxy forwards a request, `ForwardedHeadersMiddleware` will look for the `X-Forwarded-Proto` header and will update the request details as appropriate. For all subsequent middleware, the request is considered secure. When adding the middleware manually, it's important that you place `ForwardedHeadersMiddleware` before the call to `UseHsts()` or `UseHttpsRedirection()`, so that the forwarded headers are read and the request is marked secure, as appropriate.

HTTPS is one of the most basic requirements for adding security to your application these days. It can be tricky to set up initially, but once you're up and running, you can largely forget about it, especially if you're using SSL/TLS termination at a reverse proxy.

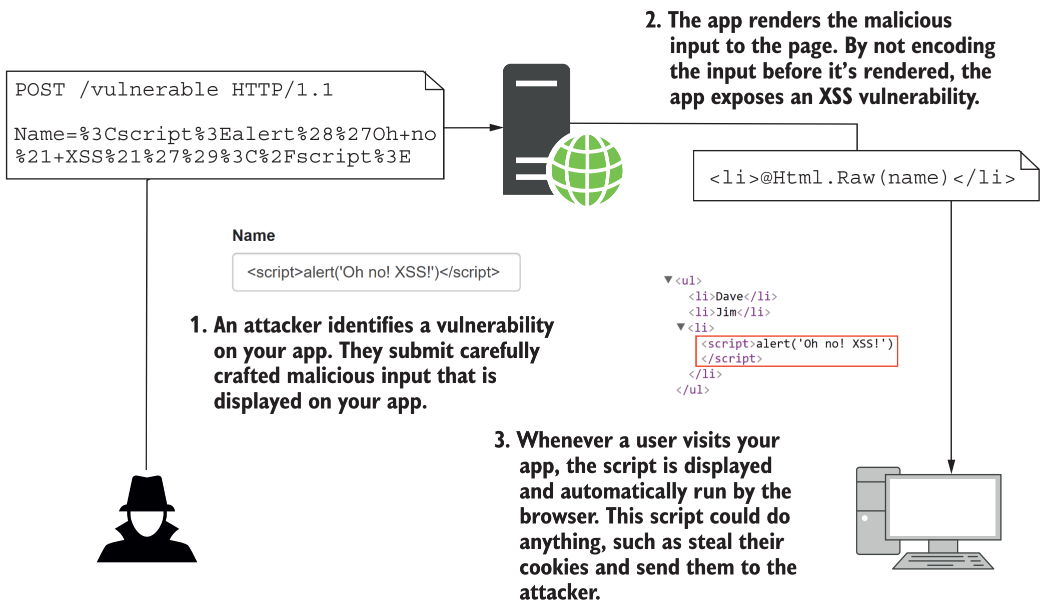
Unfortunately, most other security practices require rather more vigilance to ensure you don't accidentally introduce vulnerabilities into your app as it grows and develops. Many attacks are conceptually simple and have been known about for years, yet they're still commonly found in new applications. In the next section we'll look at one such attack and see how to defend against it when building apps using Razor Pages.

18.2 Defending against cross-site scripting (XSS) attacks

In this section I'll describe cross-site scripting attacks and how attackers can use them to compromise your users. I'll show how the Razor Pages framework protects you from these attacks, how to disable the protections when you need to, and what to look out for. I'll also discuss the difference between HTML encoding and JavaScript encoding, and the impact of using the wrong encoder.

Attackers can exploit a vulnerability in your app to create cross-site scripting (XSS) attacks that execute code in another user's browser.[6] Commonly, attackers submit content using a legitimate approach, such as an input form, which is later rendered somewhere to the page. By carefully crafting malicious input, the attacker can execute arbitrary JavaScript on a user's browser and so can steal cookies, impersonate the user, and generally do bad things.

Figure 18.5 shows a basic example of an XSS attack. Legitimate users of your app can send their name to your app by submitting a form. The app then adds the name to an internal list and renders the whole list to the page. If the names are not rendered safely, a malicious user can execute JavaScript in the browser of every other user that views the list.



```
POST /vulnerable HTTP/1.1

Name=%3Cscript%3Ealert%28%27Oh+no
%21+XSS%21%27%29%3C%2Fscript%3E
```

1. An attacker identifies a vulnerability on your app. They submit carefully crafted malicious input that is displayed on your app.

2. The app renders the malicious input to the page. By not encoding the input before it's rendered, the app exposes an XSS vulnerability.

```
<li>@Html.Raw(name)</li>
```

```
▼<ul>
   <li>Dave</li>
   <li>Jim</li>
  ▼<li>
     <script>alert('Oh no! XSS!')
     </script>
   </li>
 </ul>
```

3. Whenever a user visits your app, the script is displayed and automatically run by the browser. This script could do anything, such as steal their cookies and send them to the attacker.

Figure 18.5 How an XSS vulnerability is exploited. An attacker submits malicious content to your app, which is displayed in the browsers of other users. If the app doesn't encode the content when writing to the page, the input becomes part of the HTML of the page and can run arbitrary JavaScript.

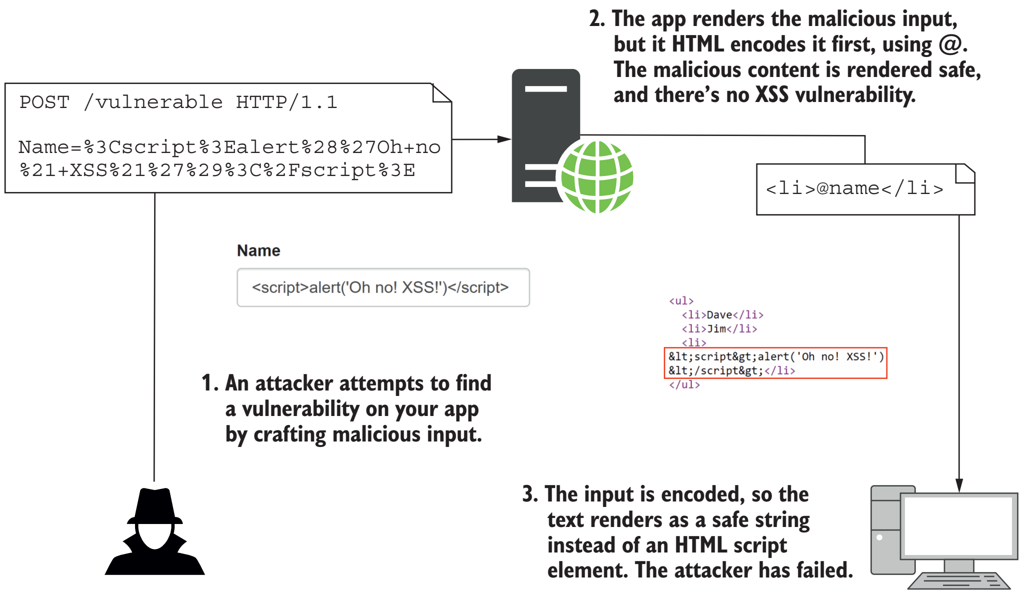
In figure 18.5 the user entered a snippet of HTML such as their name. When users view the list of names, the Razor template renders the names using `@Html.Raw()`, which writes the `<script>` tag directly to the document. The user's input has become part of the page's HTML structure. As soon as the page is loaded in a user's browser,

[6] For a detailed discussion of XSS attacks, see the "Cross Site Scripting (XSS)" article on the OWASP site: https://owasp.org/www-community/attacks/xss/.

the `<script>` tag executes, and the user is compromised. Once an attacker can execute arbitrary JavaScript on a user's browser, they can do pretty much anything.

The vulnerability here is due to rendering the user input in an unsafe way. If the data isn't encoded to make it safe before it's rendered, you could open your users to attack. *By default, Razor protects against XSS attacks* by HTML-encoding any data written using Tag Helpers, HTML Helpers, or the `@` syntax. So, generally, you should be safe, as you saw in chapter 7.

Using `@Html.Raw()` is where the danger lies—if the HTML you're rendering contains user input (even indirectly), you could have an XSS vulnerability. By rendering the user input with `@` instead, the content is encoded before it's written to the output, as shown in figure 18.6.



```
POST /vulnerable HTTP/1.1

Name=%3Cscript%3Ealert%28%27Oh+no
%21+XSS%21%27%29%3C%2Fscript%3E
```

```
<li>@name</li>
```

Name

<script>alert('Oh no! XSS!')</script>

```
<ul>
  <li>Dave</li>
  <li>Jim</li>
  <li>
&lt;script&gt;alert('Oh no! XSS!')
&lt;/script&gt;</li>
</ul>
```

2. The app renders the malicious input, but it HTML encodes it first, using @. The malicious content is rendered safe, and there's no XSS vulnerability.
1. An attacker attempts to find a vulnerability on your app by crafting malicious input.
3. The input is encoded, so the text renders as a safe string instead of an HTML script element. The attacker has failed.

Figure 18.6 Protecting against XSS attacks by HTML-encoding user input using `@` in Razor templates. The `<script>` tag is encoded so that it is no longer rendered as HTML and can't be used to compromise your app.

This example demonstrates using HTML encoding to prevent elements being directly added to the HTML DOM, but it's not the only case you have to think about. If you're passing untrusted data to JavaScript, or using untrusted data in URL query values, you must make sure you encode the data correctly.

A common scenario is when you're using jQuery or JavaScript with Razor pages, and you want to pass a value from the server to the client. If you use the standard `@` symbol to render the data to the page, the output will be HTML-encoded. Unfortunately, if you HTML-encode a string and inject it directly into JavaScript, you probably won't get what you expect.

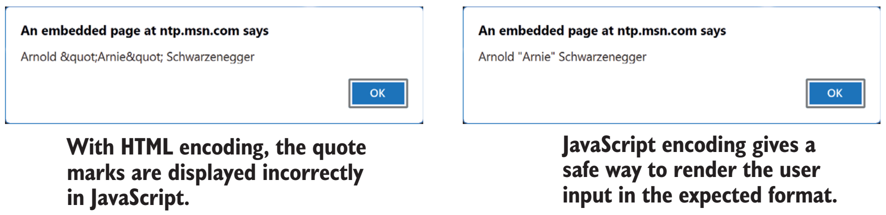
For example, if you have a variable in your Razor file called `name`, and you want to make it available in JavaScript, you might be tempted to use something like this:

```
<script>var name = '@name'</script>
```

If the name contains special characters, Razor will encode them using HTML encoding, which probably isn't what you want in this JavaScript context. For example, if `name` was `Arnold "Arnie" Schwarzenegger`, then rendering it as you did previously would give this:

```
<script>var name = 'Arnold &quot;Arnie&quot; Schwarzenegger';</script>
```

Note how the double quotation marks (`"`) have been HTML-encoded to `&quot;`. If you use this value in JavaScript directly, expecting it to be a "safe" encoded value, it's going to look wrong, as shown in figure 18.7.


An embedded page at ntp.msn.com says

Arnold &quot;Arnie&quot; Schwarzenegger

An embedded page at ntp.msn.com says

Arnold "Arnie" Schwarzenegger

OK
OK
With HTML encoding, the quote marks are displayed incorrectly in JavaScript.
JavaScript encoding gives a safe way to render the user input in the expected format.

Figure 18.7 Comparison of alerts when using JavaScript encoding compared to HTML encoding

Instead, you should encode the variable using JavaScript encoding so that the double-quote character rendered as a safe Unicode character, `\u0022`. You can achieve this by injecting a `JavaScriptEncoder` into the view (as you saw in chapter 10) and calling `Encode()` on the `name` variable:

```
@inject System.Text.Encodings.Web.JavaScriptEncoder encoder;
<script>var name = '@encoder.Encode(name)'</script>
```

To avoid having to remember to use JavaScript encoding, I recommend you don't write values into JavaScript like this. Instead, write the value to an HTML element's attributes, and then read that into the JavaScript variable later. That avoids the need for the JavaScript encoder entirely.

Listing 18.4 Passing values to JavaScript by writing them to HTML attributes

Gets a reference to the HTML element

Write the value you want in JavaScript to a data-* attribute. This will HTML-encode the data.

```
<div id="data" data-name="@name"></div>
<script>
var ele = document.getElementById('data');
var name = ele.getAttribute('data-name');     <-- Reads the data-* attribute into
</script>                                          JavaScript, which will convert it
                                                   to JavaScript encoding
```

XSS attacks are still common, and it's easy to expose yourself to them whenever you allow users to input data. Validation of the incoming data can sometimes help, but it's often a tricky problem. For example, a naive name validator might require that you only use letters, which would prevent most attacks. Unfortunately, that doesn't account for users with hyphens or apostrophes in their name, let alone users with non-western names. People get (understandably) upset when you tell them their name is invalid, so be wary of this approach!

Whether or not you use strict validation, you should always encode the data when you render it to the page. Think carefully whenever you find yourself writing `@Html.Raw()`. Is there any way for a user to get malicious data into that field? If so, you'll need to find another way to display the data.

XSS vulnerabilities allow attackers to execute JavaScript on a user's browser. The next vulnerability we're going to consider lets them make requests to your API as though they're a different logged-in user, even when the user isn't using your app. Scared? I hope so!

18.3 Protecting from cross-site request forgery (CSRF) attacks

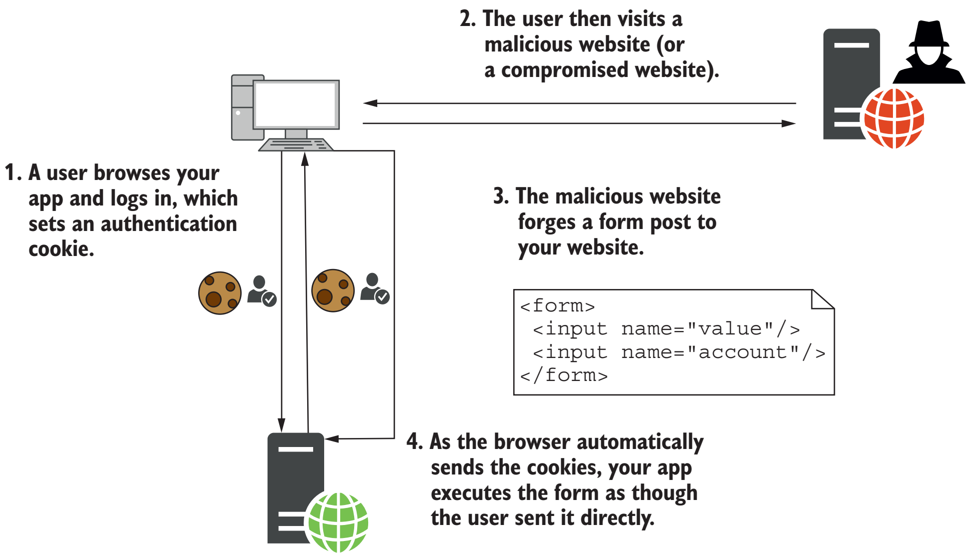
In this section you'll learn about cross-site request forgery attacks, how attackers can use them to impersonate a user on your site, and how to protect against them using anti-forgery tokens. Razor Pages protects you from these attacks by default, but you can disable these verifications, so it's important to understand the implications of doing so.

Cross-site request forgery (CSRF) attacks can be a problem for websites or APIs that use cookies for authentication. A CSRF attack involves a malicious website making an authenticated request to your API on behalf of the user, without the user initiating the request. In this section we'll explore how these attacks work and how you can mitigate them with anti-forgery tokens.

The canonical example of this attack is a bank transfer/withdrawal. Imagine you have a banking application that stores authentication tokens in a cookie, as is common (especially in traditional server-side rendered applications). Browsers automatically send the cookies associated with a domain with every request, so the app knows whether a user is authenticated.

Now imagine your application has a page that lets a user transfer funds from their account to another account using a `POST` request to the `Balance` Razor Page. You have to be logged in to access the form (you've protected the Razor Page with the `[Authorize]` attribute), but otherwise you just post a form that says how much you want to transfer, and where you want to transfer it.

Suppose a user visits your site, logs in, and performs a transaction. They then visit a second website that the attacker has control of. The attacker has embedded a form on their website that performs a POST to your bank's website, identical to the transfer funds form on your banking website. This form does something malicious, such as transfer all the user's funds to the attacker, as shown in figure 18.8. Browsers automatically send the cookies for the application when the page does a full form post, and the banking app has no way of knowing that this is a malicious request. The unsuspecting user has given all their money to the attacker!


2. The user then visits a malicious website (or a compromised website).
1. A user browses your app and logs in, which sets an authentication cookie.
3. The malicious website forges a form post to your website.
4. As the browser automatically sends the cookies, your app executes the form as though the user sent it directly.

```
<form>
 <input name="value"/>
 <input name="account"/>
</form>
```

Figure 18.8 A CSRF attack occurs when a logged-in user visits a malicious site. The malicious site crafts a form that matches one on your app and POSTs it to your app. The browser sends the authentication cookie automatically, so your app sees the request as a valid request from the user.

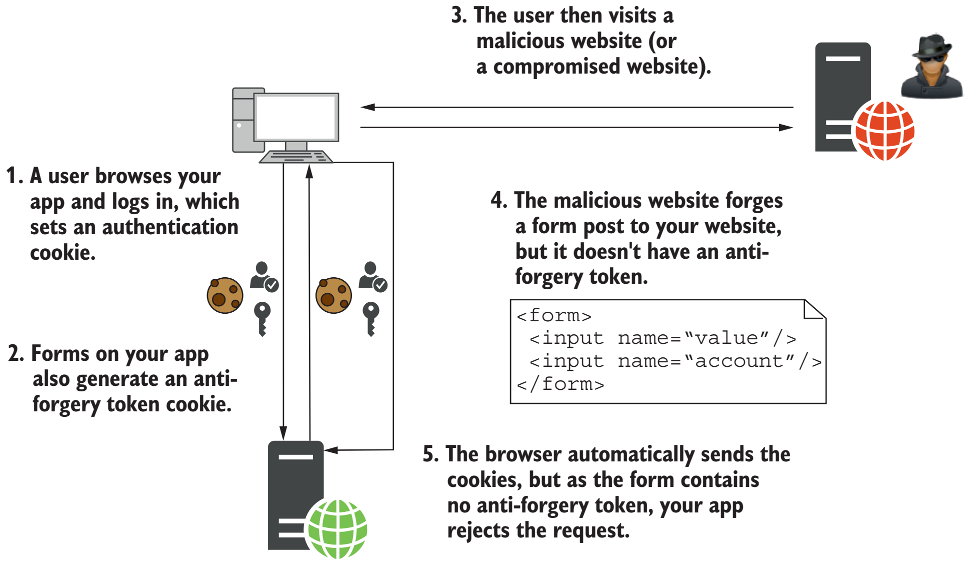
The vulnerability here revolves around the fact that browsers automatically send cookies when a page is requested (using a GET request) or a form is POSTed. There's no difference between a legitimate POST of the form in your banking app and the attacker's malicious POST. Unfortunately, this behavior is baked into the web; it's what allows you to navigate websites seamlessly after initially logging in.

A common solution to the attack is the *synchronizer token* pattern, which uses user-specific, unique, anti-forgery tokens to enforce a difference between a legitimate POST and a forged POST from an attacker.[7] One token is stored in a cookie and another is added to the form you wish to protect. Your app generates the tokens at runtime

[7] The "Cross-Site Request Forgery Prevention Cheat Sheet" article on the OWASP site gives a thorough discussion of the CSRF vulnerability, including the synchronizer token pattern: http://mng.bz/5jRa.

based on the current logged-in user, so there's no way for an attacker to create one for their forged form.

When the `Balance` Razor Page receives a form `POST`, it compares the value in the form with the value in the cookie. If either value is missing, or they don't match, the request is rejected. If an attacker creates a `POST`, the browser will post the cookie token as usual, but there won't be a token in the form itself, or the token won't be valid. The Razor Page will reject the request, protecting from the CSRF attack, as in figure 18.9.


3. The user then visits a malicious website (or a compromised website).
1. A user browses your app and logs in, which sets an authentication cookie.
4. The malicious website forges a form post to your website, but it doesn't have an anti-forgery token.
2. Forms on your app also generate an anti-forgery token cookie.
5. The browser automatically sends the cookies, but as the form contains no anti-forgery token, your app rejects the request.

```
<form>
 <input name="value"/>
 <input name="account"/>
</form>
```

Figure 18.9 Protecting against a CSRF attack using anti-forgery tokens. The browser automatically forwards the cookie token, but the malicious site can't read it, and so can't include a token in the form. The app rejects the malicious request because the tokens don't match.

The good news is that Razor Pages automatically protects you against CSRF attacks. The Form Tag Helper automatically sets an anti-forgery token cookie and renders the token to a hidden field called `__RequestVerificationToken` for every `<form>` element in your app (unless you specifically disable them). For example, take this simple Razor template that posts back to the same Razor Page:

```
<form method="post">
    <label>Amount</label>
    <input type="number" name="amount" />
    <button type="submit">Withdraw funds</button>
</form>
```

When rendered to HTML, the anti-forgery token is stored in the hidden field and is posted back with a legitimate request:

```
<form method="post">
    <label>Amount</label>
    <input type="number" name="amount" />
    <button type="submit" >Withdraw funds</button>
    <input name="__RequestVerificationToken" type="hidden"
        value="CfDJ8Daz26qb0hBGsw7QCK"/>
</form>
```

ASP.NET Core automatically adds the anti-forgery tokens to every form, and Razor Pages automatically validates them. The framework ensures the anti-forgery tokens exist in both the cookie and the form data, ensures that they match, and will reject any requests where they don't.

If you're using MVC controllers with views instead of Razor Pages, ASP.NET Core still adds the anti-forgery tokens to every form. Unfortunately, it *doesn't* validate them for you. Instead, you have to decorate your controllers and actions with `[ValidateAntiForgeryToken]` attributes. This ensures that the anti-forgery tokens exist in both the cookie and the form data, checks that they match, and rejects any requests where they don't.

> **WARNING** ASP.NET Core *doesn't* automatically validate anti-forgery tokens if you're using MVC controllers with Views. You must make sure you mark all vulnerable methods with `[ValidateAntiForgeryToken]` attributes instead, as described in the "Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core" documentation: http://mng.bz/Xd6E. Note that if you're using Web API controllers and are *not* using cookies for authentication, you are not vulnerable to CSRF attacks.

Generally, you only need to use anti-forgery tokens for `POST`, `DELETE`, and other dangerous request types that are used for modifying state; `GET` requests shouldn't be used for this purpose, so the framework doesn't require valid anti-forgery tokens to call them. Razor Pages validates anti-forgery tokens for dangerous verbs like `POST` and ignores safe verbs like `GET`. As long as you create your app following this pattern (and you should!), the framework will do the right thing to keep you safe.

If you need to explicitly ignore anti-forgery tokens on a Razor Page for some reason, you can disable the validation by applying the `[IgnoreAntiforgeryToken]` attribute to a Razor Page's `PageModel`. This bypasses the framework protections for those cases where you're doing something that you know is safe and doesn't need protecting, but in most cases it's better to play it safe and validate.

CSRF attacks can be a tricky thing to get your head around from a technical point of view, but for the most part everything *should* work without much effort on your part. Razor will add anti-forgery tokens to your forms, and the Razor Pages framework will take care of validation for you.

Where things get trickier is if you're making a lot of requests to an API using JavaScript, and you're posting JSON objects rather than form data. In these cases, you won't be able to send the verification token as part of a form (because you're sending JSON), so you'll need to add it as a header in the request instead.[8]

TIP If you're not using cookie authentication, and instead have an SPA that sends authentication tokens in a header, then good news—you don't have to worry about CSRF at all! Malicious sites can only send cookies, not headers, to your API, so they can't make authenticated requests.

Generating unique tokens with the data protection APIs

The anti-forgery tokens used to prevent CSRF attacks rely on the ability of the framework to use strong symmetric encryption to encrypt and decrypt data. Encryption algorithms typically rely on one or more keys, which are used to initialize the encryption and to make the process reproducible. If you have the key, you can encrypt and decrypt data; without it, the data is secure.

In ASP.NET Core, encryption is handled by the data protection APIs. They're used to create the anti-forgery tokens, to encrypt authentication cookies, and to generate secure tokens in general. Crucially, they also control the management of the *key files* that are used for encryption.

A key file is a small XML file that contains the random key value used for encryption in ASP.NET Core apps. It's critical that it's stored securely—if an attacker got hold of it, they could impersonate any user of your app and generally do bad things!

The data protection system stores the keys in a safe location, depending on how and where you host your app. For example,

- *Azure Web App*—In a special synced folder, shared between regions
- *IIS without user profile*—Encrypted in the registry
- *Account with user profile*—In %LOCALAPPDATA%\ASP.NET\DataProtection-Keys on Windows, or ~/.aspnet/DataProtection-Keys on Linux or macOS
- *All other cases*—In memory; when the app restarts, the keys will be lost

So why do you care? In order for your app to be able to read your users' authentication cookies, it must decrypt them using the same key that was used to encrypt them. If you're running in a web-farm scenario, then, by default, each server will have its own key and won't be able to read cookies encrypted by other servers.

[8] Exactly how you do this varies depending on the JavaScript framework you're using. Microsoft's documentation ("Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core") contains examples using JQuery and AngularJS, but you should be able to extend this to your JavaScript framework of choice: http://mng.bz/54Sl.

To get around this, you must configure your app to store its data protection keys in a central location. This could be a shared folder on a hard drive, a Redis instance, or an Azure blob storage instance, for example.

Microsoft's documentation on the data protection APIs is extremely detailed, but it can be overwhelming. I recommend reading the section on configuring data protection, ("Configure ASP.NET Core Data Protection," http://mng.bz/d40i) and configuring a key storage provider for use in a web-farm scenario ("Key storage providers in ASP.NET Core," http://mng.bz/5pW6).

It's worth clarifying that the CSRF vulnerability discussed in this section requires that a malicious site does a full form `POST` to your app. The malicious site can't make the request to your API using *client-side only* JavaScript, as browsers will block JavaScript requests to your API that are from a different origin.

This is a safety feature, but it can often cause you problems. If you're building a client-side SPA, or even if you have a little JavaScript on an otherwise server-side rendered app, you may find you need to make such *cross-origin* requests. In the next section I'll describe a common scenario you're likely to run into and show how you can modify your apps to work around it.

18.4 Calling your web APIs from other domains using CORS

In this section you'll learn about cross-origin resource sharing (CORS), a protocol to allow JavaScript to make requests from one domain to another. CORS is a frequent area of confusion for many developers, so this section describes why it's necessary and how CORS headers work. You'll then learn how to add CORS to both your whole application and specific Web API actions, and how to configure multiple CORS policies for your application.

As you've already seen, CSRF attacks can be powerful, but they would be even more dangerous if it weren't for browsers implementing the *same-origin* policy. This policy blocks apps from using JavaScript to call a web API at a different location unless the web API explicitly allows it.

DEFINITION *Origins* are deemed the same if they match the scheme (HTTP or HTTPS), domain (example.com), *and* port (80 by default for HTTP, and 443 for HTTPS). If an app attempts to access a resource using JavaScript and the origins aren't identical, the browser blocks the request.

The same-origin policy is strict—the origins of the two URLs must be identical for the request to be allowed. For example, the following origins are the same:

- http://example.com/home
- http://example.com/site.css

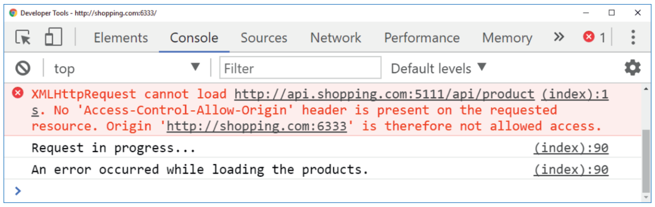
The paths are different for these two URLs (/home and /site.css), but the scheme, domain, and port (80) are identical. So if you were on the home page of your app, you could request the /site.css file using JavaScript without any issues.

In contrast, the origins of the following sites are all different, so you couldn't request any of these URLs using JavaScript from the http://example.com origin:

- *https://example.com*—Different scheme (https)
- *http://www.example.com*—Different domain (includes a subdomain)
- *http://example.com:5000*—Different port (default HTTP port is 80)

For simple apps, where you have a single web app handling all of your functionality, this limitation might not be a problem, but it's extremely common for an app to make requests to another domain.

For example, you might have an e-commerce site hosted at http://shopping.com, and you're attempting to load data from http://api.shopping.com to display details about the products available for sale. With this configuration, you'll fall foul of the same-origin policy. Any attempt to make a request using JavaScript to the API domain will fail, with an error similar to figure 18.10.



```
XMLHttpRequest cannot load http://api.shopping.com:5111/api/product (index):1
s. No 'Access-Control-Allow-Origin' header is present on the requested
resource. Origin 'http://shopping.com:6333' is therefore not allowed access.
Request in progress...                                               (index):90
An error occurred while loading the products.                        (index):90
>
```

The browser won't allow cross-origin requests by default and will block your app from accessing the response.

Figure 18.10 The console log for a failed cross-origin request. Chrome has blocked a cross-origin request from the app http://shopping.com:6333 to the API at http://api.shopping.com:5111.

The need to make cross-origin requests from JavaScript is increasingly common with the rise of client-side SPAs and the move away from monolithic apps. Luckily, there's a web standard that lets you work around this in a safe way; this standard is called cross-origin resource sharing (CORS). You can use CORS to control which apps can call your API, so you can enable scenarios like the one just described.

18.4.1 Understanding CORS and how it works

CORS is a web standard that allows your Web API to make statements about who can make cross-origin requests to it. For example, you could make statements such as these:

- Allow cross-origin requests from http://shopping.com and https://app.shopping.com.
- Only allow `GET` cross-origin requests.

- Allow returning the `Server` header in responses to cross-origin requests.
- Allow credentials (such as authentication cookies or authorization headers) to be sent with cross-origin requests.

You can combine these rules into a *policy* and apply different policies to different endpoints of your API. You could apply a policy to your entire application, or a different policy to every API action.

CORS works using HTTP headers. When your Web API application receives a request, it sets special headers on the response to indicate whether cross-origin requests are allowed, which origins they're allowed from, and which HTTP verbs and headers the request can use—pretty much everything about the request.

In some cases, before sending a real request to your API, the browser sends a *preflight* request. This is a request sent using the `OPTIONS` verb, which the browser uses to check whether it's allowed to make the real request. If the API sends back the correct headers, the browser will send the true cross-origin request, as shown in figure 18.11.

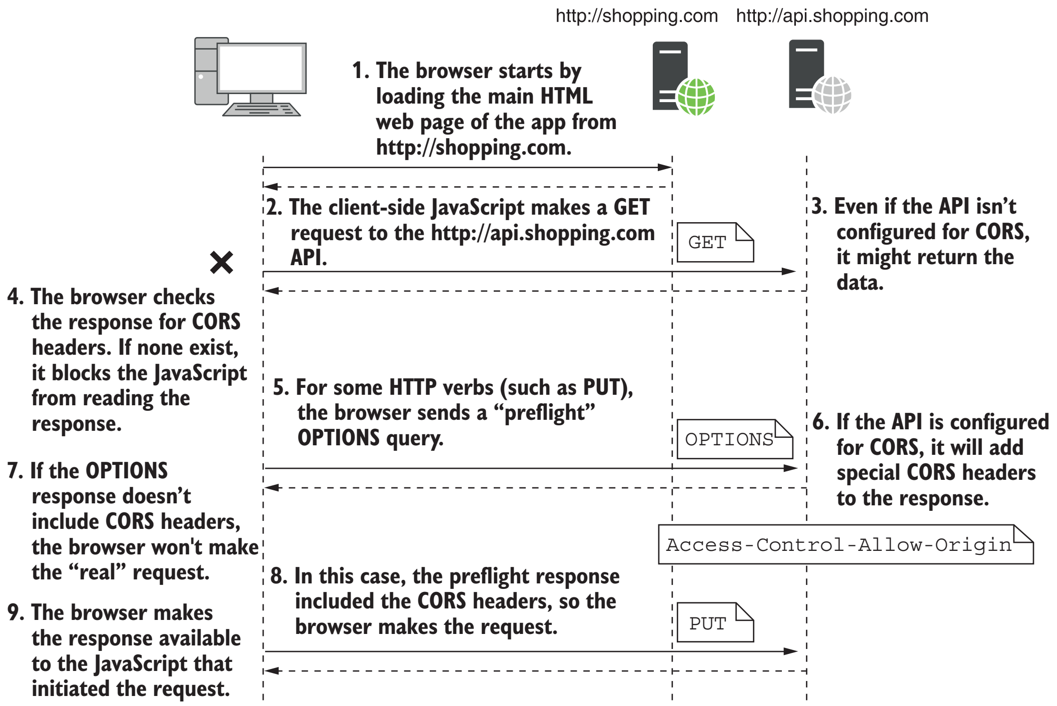

http://shopping.com
http://api.shopping.com
1. The browser starts by loading the main HTML web page of the app from http://shopping.com.
2. The client-side JavaScript makes a GET request to the http://api.shopping.com API.
GET
3. Even if the API isn't configured for CORS, it might return the data.
4. The browser checks the response for CORS headers. If none exist, it blocks the JavaScript from reading the response.
5. For some HTTP verbs (such as PUT), the browser sends a "preflight" OPTIONS query.
OPTIONS
6. If the API is configured for CORS, it will add special CORS headers to the response.
7. If the OPTIONS response doesn't include CORS headers, the browser won't make the "real" request.
Access-Control-Allow-Origin
8. In this case, the preflight response included the CORS headers, so the browser makes the request.
PUT
9. The browser makes the response available to the JavaScript that initiated the request.

Figure 18.11 Two cross-origin requests. The response to the first response doesn't contain any CORS headers, so the browser blocks the app from reading it. The second request requires a preflight `OPTIONS` request, to check if CORS is enabled. As the response contains CORS headers, the real request can be made and the response provided to the JavaScript app.

TIP For a more detailed discussion of CORS, see *CORS in Action* by Monsur Hossain (Manning, 2014), available at http://mng.bz/aD41.

The CORS specification, like many technical documents, is pretty complicated, with a variety of headers and processes to contend with.[9] Thankfully, ASP.NET Core handles the details of the specification for you, so your main concern is working out exactly *who* needs to access your API, and under what circumstances.

18.4.2 Adding a global CORS policy to your whole app

Typically, you shouldn't set up CORS for your APIs until you need it. Browsers block cross-origin communication for a reason—it closes an avenue of attack—they're not being awkward. Wait until you have an API on a different domain than an app that needs to access it.

Adding CORS support to your application requires four things:

- Add the CORS services to your app.
- Configure at least one CORS policy.
- Add the CORS middleware to your middleware pipeline.
- Either set a default CORS policy for your entire app or decorate your Web API actions with the `[EnableCors]` attribute to selectively enable CORS for specific endpoints.

Adding the CORS services to your application involves calling `AddCors()` in your `Startup.ConfigureServices` method:

```
services.AddCors();
```

The bulk of your effort in configuring CORS will go into policy configuration. A CORS policy controls how your application will respond to cross-origin requests. It defines which origins are allowed, which headers to return, which HTTP methods to allow, and so on. You normally define your policies inline when you add the CORS services to your application.

For example, consider the previous e-commerce site example. You want your API that is hosted at http://api.shopping.com to be available from the main app via client-side JavaScript, hosted at http://shopping.com. You therefore need to configure the *API* to allow cross-origin requests.

NOTE Remember, it's the main app that will get errors when attempting to make cross-origin requests, but it's the *API* you're accessing that you need to add CORS to, *not* the app making the requests.

The following listing shows how to configure a policy called `"AllowShoppingApp"` to enable cross-origin requests from http://shopping.com to the API. Additionally, we

[9] If that's the sort of thing that floats your boat, you can read the spec here: https://fetch.spec.whatwg.org/#http-cors-protocol.

explicitly allow any HTTP verb type; without this call, only simple methods (`GET`, `HEAD`, and `POST`) are allowed. The policies are built up using the familiar fluent builder style you've seen throughout this book.

Listing 18.5 Configuring a CORS policy to allow requests from a specific origin

The AddCors method exposes an Action<CorsOptions> overload.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options => {
        options.AddPolicy("AllowShoppingApp", policy =>
            policy.WithOrigins("http://shopping.com")
                .AllowAnyMethod());
    });
    // other service configuration
}
```

Every policy has a unique name.

The WithOrigins method specifies which origins are allowed. Note that the URL has no trailing /.

Allows all HTTP verbs to call the API

WARNING When listing origins in `WithOrigins()`, ensure that they don't have a trailing `"/"`; otherwise the origin will never match and your cross-origin requests will fail.

Once you've defined a CORS policy, you can apply it to your application. In the following listing, you apply the `"AllowShoppingApp"` policy to the whole application using `CorsMiddleware` by calling `UseCors()` in the `Configure` method of Startup.cs.

Listing 18.6 Adding the CORS middleware and configuring a default CORS policy

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();

    app.UseCors("AllowShoppingApp");
    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

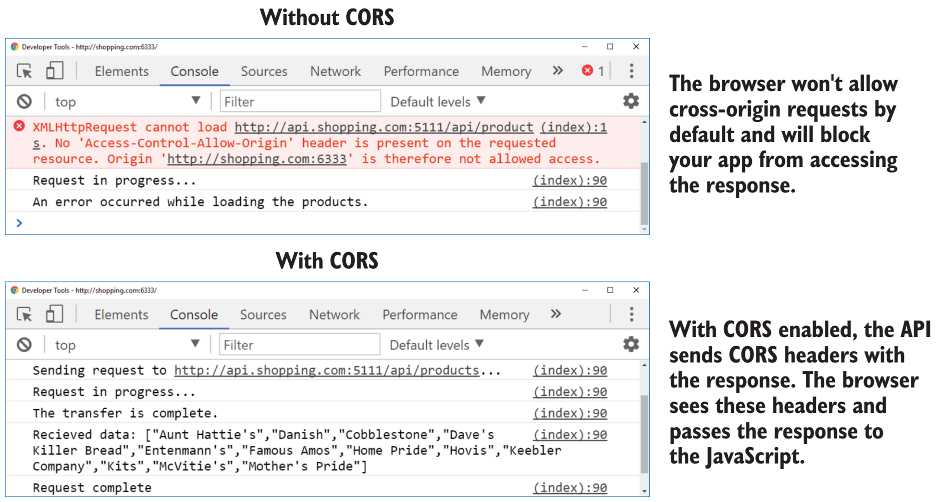
The CORS middleware must come after the call to UseRouting().

Adds the CORS middleware and uses AllowShoppingApp as the default policy

Place the CORS middleware before the endpoint middleware.

NOTE As with all middleware, the order of the CORS middleware is important. You must place the call to `UseCors()` *after* `UseRouting()` and before `UseEndpoints()`. The CORS middleware needs to intercept cross-origin requests to your Web API actions so it can generate the correct responses to preflight requests and add the necessary headers. It's typical to place the CORS middleware before the call to `UseAuthentication()`.

With the CORS middleware in place for the API, the shopping app can now make cross-origin requests. You can call the API from the http://shopping.com site and the browser lets the CORS request through, as shown in figure 18.12. If you make the same request from a domain other than http://shopping.com, the request continues to be blocked.



Without CORS

```
XMLHttpRequest cannot load http://api.shopping.com:5111/api/products. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://shopping.com:6333' is therefore not allowed access.   (index):1
Request in progress...                                        (index):90
An error occurred while loading the products.                 (index):90
```

The browser won't allow cross-origin requests by default and will block your app from accessing the response.

With CORS

```
Sending request to http://api.shopping.com:5111/api/products...  (index):90
Request in progress...                                        (index):90
The transfer is complete.                                     (index):90
Recieved data: ["Aunt Hattie's","Danish","Cobblestone","Dave's   (index):90
Killer Bread","Entenmann's","Famous Amos","Home Pride","Hovis","Keebler
Company","Kits","McVitie's","Mother's Pride"]
Request complete                                              (index):90
```

With CORS enabled, the API sends CORS headers with the response. The browser sees these headers and passes the response to the JavaScript.

Figure 18.12 With CORS enabled, as in the lower image, cross-origin requests can be made and the browser will make the response available to the JavaScript. Compare this to the upper image, in which the request was blocked.

Applying a CORS policy globally to your application in this way may be overkill. If there's only a subset of actions in your API that need to be accessed from other origins, then it's prudent to only enable CORS for those specific actions. This can be achieved with the `[EnableCors]` attribute.

18.4.3 *Adding CORS to specific Web API actions with EnableCorsAttribute*

Browsers block cross-origin requests by default for good reason—they have the potential to be abused by malicious or compromised sites. Enabling CORS for your entire app may not be worth the risk if you know that only a subset of actions will ever need to be accessed cross-origin.

If that's the case, it's best to only enable a CORS policy for those specific actions. ASP.NET Core provides the `[EnableCors]` attribute, which lets you select a policy to apply to a given controller or action method. This approach lets you apply different CORS policies to different action methods. For example, you could allow `GET` requests access to your entire API from the http://shopping.com domain, but only allow other HTTP verbs for a specific controller, while allowing anyone to access your product list action method.

You define these policies in ConfigureServices using AddPolicy() and giving the policy a name, as you saw in listing 18.5. However, instead of calling UseCors("Allow-ShoppingApp") as you saw in listing 18.6, you add the middleware without a default policy by calling UseCors() only.

To apply a policy to a controller or an action method, apply the [EnableCors] attribute, as shown in the following listing. An [EnableCors] attribute on an action takes precedence over an attribute on a controller, or you can use the [DisableCors] attribute to disable cross-origin access to the method entirely.

Listing 18.7 Applying the EnableCors attribute to a controller and action

The AllowAnyOrigin policy is "closer" to the action, so it takes precedence.

```
[EnableCors("AllowShoppingApp")]
public class ProductController: Controller
{
    [EnableCors("AllowAnyOrigin")
    public IActionResult GeteProducts() { /* Method */ }

    public IActionResult GeteProductPrice(int id) { /* Method */ }

    [DisableCors]
    public IActionResult DeleteProduct(int id) { /* Method */ }
}
```

Applies the AllowShoppingApp CORS policy to every action method

The DisableCors attribute disables CORS for the action method completely.

The AllowShoppingApp policy (from the controller) will be applied.

If you want to apply a CORS policy to most of your actions but want to use a different policy or disable CORS entirely for some actions, you can pass a default policy when configuring the middleware using UseCors("AllowShoppingApp"), for example. Actions decorated with [EnableCors("OtherPolicy")] will apply OtherPolicy preferentially, and actions decorated with [DisableCors] will not have CORS enabled at all.

Whether you choose to use a single, default CORS policy or multiple policies, you need to configure the CORS policies for your application in ConfigureServices. Many different options are available when configuring CORS. In the next section I'll provide an overview of the possibilities.

18.4.4 Configuring CORS policies

Browsers implement the cross-origin policy for security reasons, so you should carefully consider the implications of relaxing any of the restrictions they impose. Even if you enable cross-origin requests, you can still control what data cross-origin requests can send, and what your API will return. For example, you can configure

- The origins that may make a cross-origin request to your API
- The HTTP verbs (such as GET, POST, and DELETE) that can be used
- The headers the browser can send

- The headers that the browser can read from your app's response
- Whether the browser will send authentication credentials with the request

You define all of these options when creating a CORS policy in your call to `AddCors()` using the `CorsPolicyBuilder`, as you saw in listing 18.5. A policy can set all or none of these options, so you can customize the results to your heart's content. Table 18.1 shows some of the options available, and their effects.

Table 18.1 The methods available for configuring a CORS policy, and their effect on the policy

`CorsPolicyBuilder` method example	Result
`WithOrigins("http://shopping.com")`	Allows cross-origin requests from http://shopping.com.
`AllowAnyOrigin()`	Allows cross-origin requests from any origin. This means *any* website can make JavaScript requests to your API.
`WithMethods()/AllowAnyMethod()`	Sets the allowed methods (such as `GET`, `POST`, and `DELETE`) that can be made to your API.
`WithHeaders()/AllowAnyHeader()`	Sets the headers that the browser may send to your API. If you restrict the headers, you must include at least `"Accept"`, `"Content-Type"`, and `"Origin"` to allow valid requests.
`WithExposedHeaders()`	Allows your API to send extra headers to the browser. By default, only the `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified`, and `Pragma` headers are sent in the response.
`AllowCredentials()`	By default, the browser won't send authentication details with cross-origin requests unless you explicitly allow it. You must also enable sending credentials client-side in JavaScript when making the request.

One of the first issues in setting up CORS is realizing you have a cross-origin problem at all. Several times I've been stumped trying to figure out why a request won't work, until I realize the request is going cross-domain, or from HTTP to HTTPS, for example.

Whenever possible, I recommend avoiding cross-origin requests completely. You can end up with subtle differences in the way browsers handle them, which can cause more headaches. In particular, avoid HTTP to HTTPS cross-domain issues by running all of your applications behind HTTPS. As discussed in section 18.1, that's a best practice anyway, and it'll help avoid a whole class of CORS headaches.

Once I've established I definitely need a CORS policy, I typically start with the `WithOrigins()` method. I then expand or restrict the policy further, as need be, to provide cross-origin lockdown of my API, while still allowing the required functionality. CORS can be tricky to work around, but remember, the restrictions are there for your safety.[10]

Cross-origin requests are only one of many potential avenues attackers could use to compromise your app. Many of these are trivial to defend against, but you need to be aware of them, and know how to mitigate them. In the next section we'll look at common threats and how to avoid them.

18.5 Exploring other attack vectors

So far in this chapter, I've described two potential ways attackers can compromise your apps—XSS and CSRF attacks—and how to prevent them. Both of these vulnerabilities regularly appear on the OWASP top ten list of most critical web app risks,[11] so it's important to be aware of them and to avoid introducing them into your apps. In this section I'll provide an overview of some of the other most common vulnerabilities and how to avoid them in your apps.

18.5.1 Detecting and avoiding open redirect attacks

A common OWASP vulnerability is due to *open redirect* attacks. An open redirect attack is where a user clicks a link to an otherwise safe app and ends up being redirected to a malicious website, such as one that serves malware. The safe app contains no direct links to the malicious website, so how does this happen?

Open redirect attacks occur where the next page is passed as a parameter to an action method. The most common example is when you're logging in to an app. Typically, apps remember the page a user is on before redirecting them to a login page by passing the current page as a `returnUrl` query string parameter. After the user logs in, the app redirects the user to the `returnUrl` to carry on where they left off.

Imagine a user is browsing an e-commerce site. They click Buy on a product and are redirected to the login page. The product page they were on is passed as the `returnUrl`, so after they log in, they're redirected to the product page instead of being dumped back to the home screen.

An open redirect attack takes advantage of this common pattern, as shown in figure 18.13. A malicious attacker creates a login URL where the `returnUrl` is set to the website they want to send the user to and convinces the user to click the link to your web app. After the user logs in, a vulnerable app will then redirect the user to the malicious site.

The simple solution to this attack is to always validate that the `returnUrl` is a local URL that belongs to your app *before* redirecting users to it. The default Identity UI

[10] You can find the OWASP top ten list here: https://owasp.org/www-project-top-ten/.

[11] OWASP publishes the list online, with descriptions of each attack and how to prevent those attacks. There's a cheat sheet for staying safe here: https://cheatsheetseries.owasp.org/.

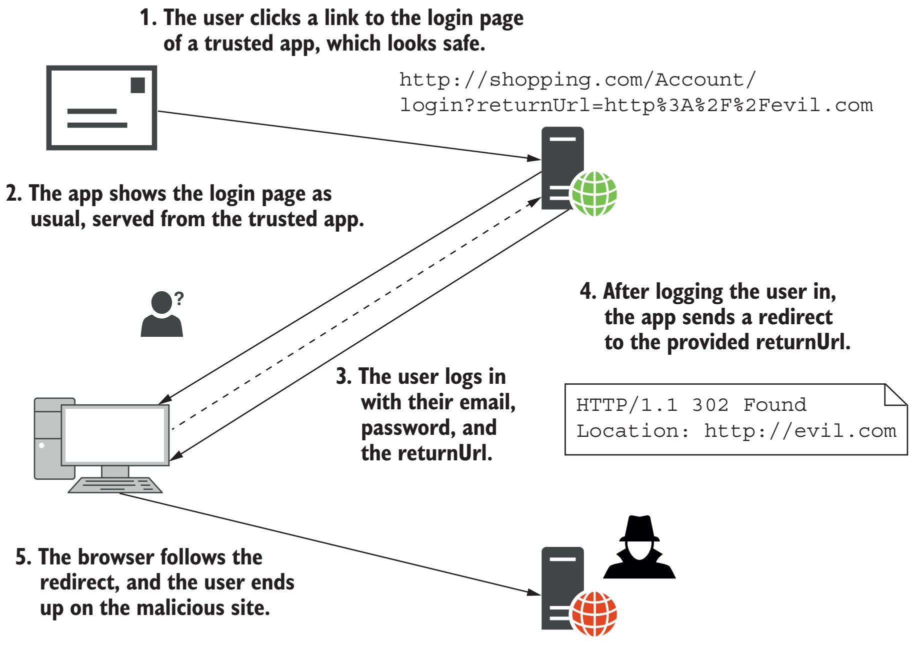

1. The user clicks a link to the login page of a trusted app, which looks safe.
http://shopping.com/Account/login?returnUrl=http%3A%2F%2Fevil.com
2. The app shows the login page as usual, served from the trusted app.
3. The user logs in with their email, password, and the returnUrl.
4. After logging the user in, the app sends a redirect to the provided returnUrl.
HTTP/1.1 302 Found
Location: http://evil.com
5. The browser follows the redirect, and the user ends up on the malicious site.

Figure 18.13 An open redirect makes use of the common return URL pattern. This is typically used for login pages but may be used in other areas of your app too. If your app doesn't verify that the URL is safe before redirecting the user, it could redirect users to malicious sites.

does this already, so you shouldn't have to worry about the login page if you're using Identity, as described in chapter 14.

If you have redirects in other parts of your app, ASP.NET Core provides a couple of helper methods for staying safe, the most useful of which is `Url.IsLocalUrl()`. The following listing shows how you could verify that a provided return URL is safe, and if it isn't, redirect to the app's homepage. You can also use the `LocalRedirect()` helper method on the `ControllerBase` and Razor Page `PageModel` classes, which throws an exception if the provided URL isn't local.

Listing 18.8 Detecting open redirect attacks by checking for local return URLs

```
[HttpPost]
public async Task<IActionResult> Login(
    LoginViewModel model, string returnUrl = null)
{
    // Verify password, and sign user in

    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
```

The return URL is provided as an argument to the action method.

Returns true if the return URL starts with / or ~/

The URL is local, so it's safe to redirect to it.

```
    else
    {
        return RedirectToAction("Index", "Home");
    }
}
```

The URL was not local and could be an open redirect attack, so redirect to the homepage for safety.

This simple pattern protects against open redirect attacks that could otherwise expose your users to malicious content. Whenever you're redirecting to a URL that comes from a query string or other user input, you should use this pattern.

Open redirect attacks present a risk to your *users* rather than to your app directly. The next vulnerability represents a critical vulnerability in your app itself.

18.5.2 Avoiding SQL injection attacks with EF Core and parameterization

SQL injection attacks represent one of the most dangerous threats to your application. Attackers craft simple malicious input, which they send to your application as traditional form-based input or by customizing URLs and query strings to execute arbitrary code against your database. An SQL injection vulnerability could expose your entire database to attackers, so it's critical that you spot and remove any such vulnerabilities in your apps.

Hopefully I've scared you a little with that introduction, so now for the good news—if you're using EF Core (or pretty much any other ORM) in a standard way, you should be safe. EF Core has built-in protections against SQL injection, so as long as you're not doing anything funky, you should be fine.

SQL injection vulnerabilities occur when you build SQL statements yourself and include dynamic input that an attacker provides, even indirectly. EF Core provides the ability to create raw SQL queries using the `FromSqlRaw()` method, so you must be careful when using this method.

Imagine your recipe app has a search form that lets you search for a recipe by name. If you write the query using LINQ extension methods (as discussed in chapter 12), then you would have no risk of SQL injection attacks. However, if you decide to write your SQL query by hand, you open yourself up to such a vulnerability.

Listing 18.9 An SQL injection vulnerability in EF Core due to string concatenation

```
public IList<User> FindRecipe(string search)
{
    return _context.Recipes
        .FromSqlRaw("SELECT * FROM Recipes" +
                  "WHERE Name = '" + search + "'")
        .ToList();
}
```

The search parameter comes from user input, so it's unsafe.

You can write queries by hand using the FromSqlRaw extension method.

This introduces the vulnerability—including unsafe content directly in an SQL string.

The current EF Core DbContext is held in the _context field.

In this listing, the user input held in `search` is included *directly* in the SQL query. By crafting malicious input, users can potentially perform any operation on your database. Imagine an attacker searches your website using the text

```
'; DROP TABLE Recipes; --
```

Your app assigns this to the `search` parameter, and the SQL query executed against your database becomes

```
SELECT * FROM Recipes WHERE Name = ''; DROP TABLE Recipes; --'
```

By simply entering text into the search form of your app, the attacker has deleted the entire Recipes table from your app! That's catastrophic, but an SQL injection vulnerability provides more or less unfettered access to your database. Even if you've set up database permissions correctly to prevent this sort of destructive action, attackers will likely be able to read all the data from your database, including your users' details.

The simple way to avoid this happening is to avoid creating SQL queries by hand like this. If you do need to write your own SQL queries, don't use string concatenation, as in listing 18.9. Instead, use parameterized queries, in which the (potentially unsafe) input data is separate from the query itself, as shown here.

Listing 18.10 Avoiding SQL injection by using parameterization

```
public IList<User> FindRecipe(string search)
{
    return _context.Recipes
        .FromSqlRaw("SELECT * FROM Recipes WHERE Name = '{0}'",
                search)
        .ToList();
}
```

The SQL query uses a placeholder {0} for the parameter.

The dangerous input is passed as a parameter, separate from the query.

Parameterized queries are not vulnerable to SQL injection attacks, so the attack presented earlier won't work. If you use EF Core (or other ORMs) to access data using standard LINQ queries, you won't be vulnerable to injection attacks. EF Core will automatically create all SQL queries using parameterized queries to protect you.

NOTE I've only talked about SQL injection attacks in terms of a relational database, but this vulnerability can appear in NoSQL and document databases too. Always use parameterized queries (or the equivalent), and don't craft queries by concatenating strings with user input.

Injection attacks have been the number one vulnerability on the web for over a decade, so it's crucial that you're aware of them and how they arise. Whenever you need to write raw SQL queries, make sure you always use parameterized queries.

The next vulnerability is also related to attackers accessing data they shouldn't be able to. It's a little subtler than a direct injection attack but is trivial to perform—the only skill the attacker needs is the ability to count.

18.5.3 Preventing insecure direct object references

Insecure direct object reference is a bit of a mouthful, but it means users accessing things they shouldn't by noticing patterns in URLs. Let's revisit our old friend the recipe app. As a reminder, the app shows you a list of recipes. You can view any of them, but you can only edit recipes you created yourself. When you view someone else's recipe, there's no Edit button visible.

For example, a user clicks the Edit button on one of their recipes and notices the URL is `/Recipes/Edit/120`. That "120" is a dead giveaway as the underlying database ID of the entity you're editing. A simple attack would be to change that ID to gain access to a *different* entity, one that you wouldn't normally have access to. The user could try entering `/Recipes/Edit/121`. If that lets them edit or view a recipe that they shouldn't be able to, you have an insecure direct object reference vulnerability.

The solution to this problem is simple—you should have resource-based authentication and authorization in your action methods. If a user attempts to access an entity they're not allowed to access, they should get a permission-denied error. They shouldn't be able to bypass your authorization by typing a URL directly into the search bar of their browser.

In ASP.NET Core apps, this vulnerability typically arises when you attempt to restrict users by hiding elements from your UI, such as by hiding the Edit button. Instead, you should use resource-based authorization, as discussed in chapter 15.

> **WARNING** You must always use resource-based authorization to restrict which entities a user can access. Hiding UI elements provides an improved user experience, but it isn't a security measure.

You can sidestep this vulnerability somewhat by avoiding integer IDs for your entities in the URLs; for example, using a pseudorandom GUID (for instance, `C2E296BA-7EA8-4195-9CA7-C323304CCD12`) instead. This makes the process of guessing other entities harder, as you can't just add one to an existing number, but it's only masking the problem rather than fixing it. Nevertheless, using GUIDs can be useful when you want to have publicly accessible pages (that don't require authentication), but you don't want their IDs to be easily discoverable.

The final section in this chapter doesn't deal with a single vulnerability. Instead, I'll discuss a separate, but related, issue: protecting your users' data.

18.5.4 Protecting your users' passwords and data

For many apps, the most sensitive data you'll be storing is the personal data of your users. This could include emails, passwords, address details, or payment information. You should be careful when storing any of this data. As well as presenting an inviting target for attackers, you may have legal obligations for how you handle it, such as data protection laws and PCI compliance requirements.

The easiest way to protect yourself is to not store data that you don't need. If you don't *need* your user's address, don't ask for it. That way, you can't lose it! Similarly, if you use a third-party identity service to store user details, as described in chapter 14, you won't have to work as hard to protect your users' personal information.

If you store user details in your own app, or build your own identity provider, then you need to make sure to follow best practices when handling user information. The new project templates that use ASP.NET Core Identity follow most of these practices by default, so I highly recommend you start from one of these. You need to consider many different aspects—too many to go into detail here[12]—but they include the following:

- Never store user passwords anywhere directly. You should only store cryptographic hashes, computed using an expensive hashing algorithm, such as BCrypt or PBKDF2.
- Don't store more data than you need. You should never store credit card details.
- Allow users to use two-factor authentication (2FA) to sign in to your site.
- Prevent users from using passwords that are known to be weak or compromised.
- Mark authentication cookies as "http" (so they can't be read using JavaScript) and "secure" so they'll only be sent over an HTTPS connection, never over HTTP.
- Don't expose whether a user is already registered with your app or not. Leaking this information can expose you to enumeration attacks.[13]

These are all guidelines, but they represent the minimum you should be doing to protect your users. The most important thing is to be aware of potential security issues as you're building your app. Trying to bolt on security at the end is always harder than thinking about it from the start, so it's best to think about it earlier rather than later.

This chapter has been a whistle-stop tour of things to look out for. We've touched on most of the big names in security vulnerabilities, but I strongly encourage you to check out the other resources mentioned in this chapter. They provide a more exhaustive list of things to consider, complementing the defenses mentioned in this chapter. On top of that, don't forget about input validation and mass assignment/over-posting, as discussed in chapter 6. ASP.NET Core includes basic protections against some of the most common attacks, but you can still shoot yourself in the foot. Make sure it's not your app making headlines for being breached!

Summary

- HTTPS is used to encrypt your app's data as it travels from the server to the browser and back. This prevents third parties from seeing or modifying it.
- HTTPS is virtually mandatory for production apps, as modern browsers like Chrome and Firefox mark non-HTTPS apps as explicitly "not secure."

[12] The NIST (National Institute of Standards and Technology) recently released their Digital Identity Guidelines on how to handle user details: http://mng.bz/6gRA.

[13] You can learn more about website enumeration in this video tutorial from Troy Hunt: http://mng.bz/PAAA.

- In production, you can avoid handling the TLS in your app by using SSL/TLS offloading. This is where a reverse proxy uses HTTPS to talk to the browser, but the traffic is unencrypted between your app and the reverse proxy. The reverse proxy could be on the same or a different server, such as IIS or NGINX, or it could be a third-party service, such as Cloudflare.
- You can use the ASP.NET Core developer certificate or the IIS express developer certificate to enable HTTPS during development. This can't be used for production, but it's sufficient for testing locally. You must run `dotnet dev-certs https --trust` when you first install the .NET SDK to trust the certificate.
- You can configure an HTTPS certificate for Kestrel in production using the `Kestrel:Certificates:Default` configuration section. This does not require any changes to your application—Kestrel will automatically load the certificate when your app starts and use it to serve HTTPS requests.
- You can use the `HstsMiddleware` to set HTTP Strict Transport Security (HSTS) headers for your application, to ensure the browser sends HTTPS requests to your app instead of HTTP requests. This can only be enforced once an HTTPS request is made to your app, so it's best used in conjunction with HTTP to HTTPS redirection.
- You can enforce HTTPS for your whole app using the `HttpsRedirectionMiddleware`. This will redirect HTTP requests to HTTPS endpoints.
- Cross-site scripting (XSS) attacks involve malicious users injecting content into your app, typically to run malicious JavaScript when users browse your app. You can avoid XSS injection attacks by always encoding unsafe input before writing it to a page. Razor Pages do this automatically unless you use the `@Html.Raw()` method, so use it sparingly and carefully.
- Cross-site request forgery (CSRF) attacks are a problem for apps that use cookie-based authentication, such as ASP.NET Core Identity. It relies on the fact that browsers automatically send cookies to a website. A malicious website could create a form that `POST`s to your site, and the browser will send the authentication cookie with the request. This allows malicious websites to send requests as though they're the logged-in user.
- You can mitigate CSRF attacks using anti-forgery tokens. These involve writing a hidden field in every form that contains a random string based on the current user. A similar token is stored in a cookie. A legitimate request will have both parts, but a forged request from a malicious website will only have the cookie half; they cannot recreate the hidden field in the form. By validating these tokens, your API can reject forged requests.
- The Razor Pages framework automatically adds anti-forgery tokens to any forms you create using Razor and validates the tokens for inbound requests. You can disable the validation check if necessary, using the `[IgnoreAntiForgeryToken]` attribute.

- Browsers won't allow websites to make JavaScript AJAX requests from one app to others at different origins. To match the origin, the app must have the same scheme, domain, and port. If you wish to make cross-origin requests like this, you must enable cross-origin resource sharing (CORS) in your API.
- CORS uses HTTP headers to communicate with browsers and defines which origins can call your API. In ASP.NET Core, you can define multiple policies, which can be applied either globally to your whole app, or to specific controllers and actions.
- You can add the CORS middleware by calling `UseCors()` in `Startup.Configure` and optionally providing the name of the default CORS policy to apply. You can also apply CORS to a Web API action or controller by adding the `[EnableCors]` attribute and providing the name of the policy to apply.
- Open redirect attacks use the common `returnURL` mechanism after logging in to redirect users to malicious websites. You can prevent this attack by ensuring you only redirect to local URLs—URLs that belong to your app.
- Insecure direct object references are a common problem where you expose the ID of database entities in the URL. You should always verify that users have permission to access or change the requested resource by using resource-based authorization in your action methods.
- SQL injection attacks are a common attack vector when you build SQL requests manually. Always use parameterized queries when building requests, or instead use a framework like EF Core, which isn't vulnerable to SQL injection.
- The most sensitive data in your app is often the data of your users. Mitigate this risk by only storing data that you need. Ensure you only store passwords as a hash, protect against weak or compromised passwords, and provide the option for 2FA. ASP.NET Core Identity provides all of this out of the box, so it's a great choice if you need to create an identity provider.